20-单一职责原则：你的模块到底为谁负责？

你好！我是郑晔。

经过前面的讲解，我们对各种编程范式已经有了基本的理解，也知道了自己手上有哪些可用的设计元素。但只有这些元素是不够的，我们还需要一些比编程范式更具体的内容来指导工作。从这一讲开始，我们就进入到设计原则的学习。

在众多的原则中，我们该学习哪个呢？我选择了SOLID原则，因为SOLID原则是一套比较成体系的设计原则。它不仅可以指导我们设计模块（在面向对象领域，模块主要指的就是类），还可以被当作一把尺子，来衡量我们设计的有效性。

那SOLID原则是什么呢？它实际上是五个设计原则首字母的缩写，它们分别是：

- 单一职责原则（Single responsibility principle，SRP）
- 开放封闭原则（Open–closed principle，OCP）
- Liskov替换原则（Liskov substitution principle，LSP）
- 接口隔离原则（Interface segregation principle，ISP）
- 依赖倒置原则（Dependency inversion principle，DIP）

这些设计原则是由Robert Martin提出并逐步整理和完善的。他在《敏捷软件开发：原则、实践与模式》和《架构整洁之道》两本书中，对SOLID原则进行了两次比较完整的阐述。在这两本时隔近20年的书里，你可以看到Robert Martin对SOLID原则的理解一步步在深化，如果你想了解原作者的思考，这两本书都推荐你阅读。

那么，在接下来的几讲中，我就来给你讲解这五个设计原则，除了设计原则的基本内容之外，我还会把我自己的理解增补其中，把两本书中没有讲到的一些逻辑给你补充进去。

好，我们开始，率先登场的当然就是单一职责原则。

变化的原因

单一职责原则，这个名字非常容易让我们望文生义，我们可能会理解成，一个类只干一件事，这看起来似乎是一项再合理不过的要求了。因为，几乎所有的程序员都知道“高内聚、低耦合”，都知道该把相关的代码放到一起。

所以，如果我们随便拿一个模块去问他的作者，这个模块是不是只做了一件事，他们的答案几乎都会是一样的：是的，只做了一件事。那么，既然这个设计原则如此通用，以至于所有的人都可以做到，那我们为什么还要有这样一个设计原则呢？

原因就在于，我们一开始的理解就是错的，我们把单一职责理解成了有关如何组合的原则，但实际上，单一职责是关于如何分解的。

那到底什么是单一职责原则呢？

正如Robert Martin所说，单一职责的定义经历了一些变化。在《敏捷软件开发：原则、实践与模式》中其

定义是，“一个模块应该有且仅有一个变化的原因”；而到了《架构整洁之道》中，其定义就变成了“一个模块应该对一类且仅对一类行为者（actor）负责”。

单一职责原则和一个类只干一件事之间，最大的差别就是，**将变化纳入了考量**。

我们先分析第一个定义：一个模块应该有且仅有一个变化的原因。我们在课程一开始就在说，软件设计是一门关注长期变化的学问。变化是我们最不愿意面对却不得不面对的事，因为变化会引发新的不确定性，可能是新增功能自身的稳定问题，也可能是旧有功能遭到破坏带来的问题。

所以，**一个模块最理想的状态是不改变，其次是少改变**，它可以成为一个模块设计好坏的衡量标准。

在真实项目中，一个模块之所以会频繁变化，关键点就在于能引起它改变的原因太多了。

怎么理解呢？我们来看一个例子。假设我们要开发一个项目管理的工具，自然少不了一个用户的类，我们可能设计出这样一个用户类：

```
// 用户类
class User {
  // 修改密码
  void changePassword(String password);
  // 加入一个项目
  void joinProject(Project project);
  // 接管一个项目，成为管理员
  void takeOverProject(Project project);
  ...
}
```

看上去，这个类设计得还挺合理，有用户信息管理、有项目管理等等。没过多久，新的需求来了，要求每个用户能够设置电话号码，所以，你给它增加了一个新的方法：

```
  void changePhoneNumber(PhoneNumber phoneNumber):
```

过了几天，又来了新需求，要查看一个用户加入了多少项目：

```
  int countProject();
```

就这样，左一个需求，右一个需求，几乎每个需求都要改到这个类。那会导致什么结果呢？一方面，这个类会不断膨胀；另一方面，内部的实现会越来越复杂。按照我们提出的衡量标准，这个类变动的频繁程度显然是不理想的，主要原因就在于它引起变动的需求太多了：

- 为什么要增加电话号码呢？因为这是用户管理的需求。用户管理的需求还会有很多，比如，用户实名认证、用户组织归属等等；

- 为什么要查看用户加入多少项目呢？这是项目管理的需求。项目管理的需求还会有很多，比如，团队管理、项目权限等等。

这就是两种完全不同的需求，但它们都改到了同一个类，所以，这个User类就很难稳定下来。解决这种问题，最好的办法就是把不同的需求引起的变动拆分开来。针对这里的用户管理和项目管理两种不同需求，我们完全可以把这个User类拆成两个类。比如，像下面这样，把用户管理类的需求放到User类里，把项目管理类的需求放到Member类里：

```
// 用户类
class User {
  // 修改密码
  void changePassword(String password);
  ...
}

// 项目成员类
class Member
  // 加入一个项目
  void joinProject(Project project);
  // 接管一个项目，成为管理员
  void takeOverProject(Project project);
  ...
}
```

如此一来，用户管理的需求只要调整User类就好，而项目管理的需求只要调整Member类即可，二者各自变动的理由就少了一些。

变化的来源

跟着我们课程一路学下来的同学可能发现了，上面的做法与我们之前讨论过的分离关注点很像。

确实是这样的，想要更好地理解单一职责原则，重要的就是要把不同的关注点分离出来。在上面这个例子中，分离的是不同的业务关注点。所以，**理解单一职责原则本质上就是要理解分离关注点**。

按照之前的说法，分离关注点，应该是发现的关注点越多越好，粒度越小越好。如果你能看到的关注点越多，就可以构建出更多的类，但每个类的规模相应地就会越小，与之相关的需求变动也会越少，它能够稳定下来的几率就会越大。我们代码库里**稳定的类越多越好，这应该是我们努力的一个方向**。

不过，也许你会想，如果将这种思路推演到极致，一个类应该只有一个方法，这样，它受到的影响应该是最小的。的确如此，但我们在真实项目中，一个类通常都不只有一个方法，如果我们要求所有人都做到极致，显然也是不现实的。

那应该把哪些内容组织到一起呢？这就需要我们考虑单一职责原则定义的升级版，也就是第二个定义：一个模块应该对一类且仅对一类行为者负责。

如果说第一个定义将变化纳入了考量，那这个升级版的定义则将变化的来源纳入了考量。

需求为什么会改变？因为有各种提出需求的人，不同的人提出的需求，其关注点是不同的。在前面的那个关

于用户的讨论中，关心用户管理和关心项目管理的可能就是两拨完全不同的人，至少他们在提需求的时候扮演的是两种不同的角色。

两种不同角色的人，两件不同的事，到了代码里却混在了一起，这是不合理的。所以，分开才是一个好选择。用户管理的人，我和他们聊User，项目管理的人，我们来讨论Member。

> 康威定律：一个组织设计出的系统，其结构受限于其组织的沟通结构。

Robert Martin说，单一职责原则是基于康威定律的一个推论：一个软件系统的最佳结构高度依赖于使用这个软件的组织的内部结构。如果我们的软件结构不能够与组织结构对应，就会带来一系列麻烦，前面的那个例子只是一个小例子。

实际上，当我们更新了对于单一职责原则的理解，你会发现，它的应用范围不仅仅可以放在类这样的级别，也可以放到更大的级别。

我给你举个例子。我曾经接触过一个交易平台，其中有一个关键模型：手续费率，就是交易一次按什么比例收取佣金。平台可以利用手续费率做不同的活动，比如，给一些人比较低的手续费率，鼓励他们来交易，不同的手续费率意味着对不同交易行为的鼓励。

所以，对运营人员来说，手续费率是一个可以玩出花的东西。然而，对交易系统而言，稳定高效是重点。显然，经常修改的手续费率和稳定的系统之间存在矛盾。

经过分析，我们发现，这是两类不同的行为者。所以，在设计的时候，我们把手续费率设置放到运营子系统，而交易子系统只负责读取手续费率。当运营子系统修改了手续费率，会把最新的结果更新到交易子系统中。至于各种手续费率设置的花样，交易子系统根本不需要关心。

你看，单一职责原则也可以指导我们在不同的子系统之间进行职责分配。所以，单一职责原则这个看起来最简单的原则，实际上也蕴含着很多值得挖掘的内容。要想理解好单一职责原则：

- 我们需要理解封装，知道要把什么样的内容放到一起；
- 我们需要理解分离关注点，知道要把不同的内容拆分开来；
- 我们需要理解变化的来源，知道把不同行为者负责的代码放到不同的地方。

在《10x程序员工作法》中，我也提到过单一职责原则，不过我是从自动化和任务分解的角度进行讲解的，其中讨论到了函数要小。结合今天的内容，你就可以更好地理解函数要小的含义了，每个函数承担的职责要单一，这样，它才能稳定下来。

总结时刻

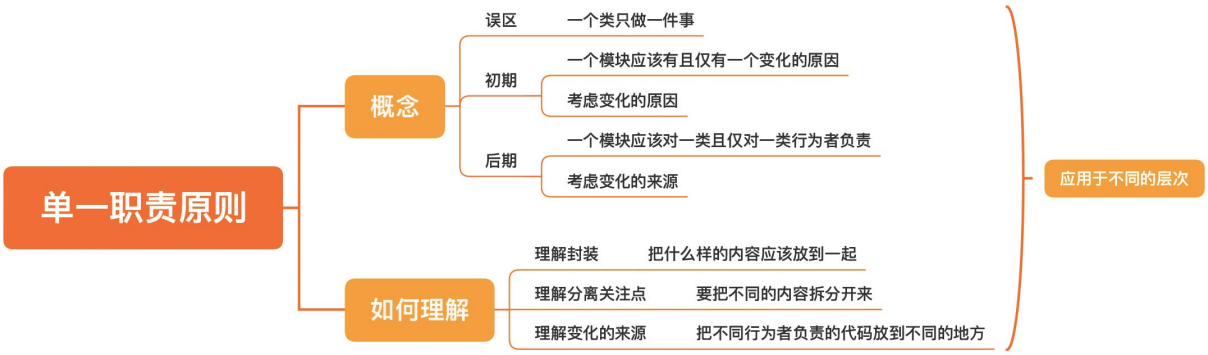
今天，我们学习了单一职责原则。单一职责原则讲的并不是一个类只做一件事，它的关注点在于变化。其最初的定义是一个模块应该有且仅有一个变化的原因，后来其定义升级为一个模块应该对一类且仅对一类行为者负责。这个定义从考虑变化升级到考虑变化的来源。

单一职责原则，本质上体现的还是分离关注点，所以，它与分离关注点的思考角度是一样的，需要我们将模块拆分成更小的粒度。不过，相比于分离关注点，它会更加具体，因为它需要我们考察关注点的来源：不同的行为者。

单一职责原则可以应用于不同的层次，小到一个函数，大到一个系统，我们都可以用它来衡量我们的设计。

好，我们已经了解了SOLID的第一个原则：单一职责原则。下一讲，我们再来看下一个原则：开放封闭原则。

如果今天的内容你只能记住一件事，那请记住：**应用单一职责原则衡量模块，粒度越小越好。**



极客时间

思考题

最后，我想请你反思一下，在你现有的系统设计中，有没有不符合单一职责原则的地方呢？应该如何改进呢？欢迎在留言区写下你的想法。

感谢阅读，如果你觉得这一讲的内容对你有帮助的话，也欢迎把它分享给你的朋友。

精选留言：

- Being 2020-07-14 09:07:01
  老师提出的三个理解，是个层层递进的过程。
  单个原子性模块固化下来的越多，可积累的就越多。

- 阳仔 2020-07-13 13:08:28
  单一职责原则，相信很多人都了解过，但为何还是会出现高度耦合，职责不明的代码逻辑？
  我觉得最根本原因是没有深入理解这个职责内涵（当然我自己也是），一开始的时候我们都自信满满的希望写出漂亮的代码，但随着版本迭代过程中，需求也不断变化，不知不觉就陷入到了变化之中。我们应该把这个原则铭记于心，当我们要修改这个模块或者类的时候，都要思考一下
  我为何要修改这个类？这部分修改可不可以放在其它地方？

  作者回复2020-07-13 15:30:41
  指望每个人铭记是不现实的，需要配合代码评审去发现问题。

- Jxin 2020-07-13 10:55:55
  1.老项目大部分应该都没有规范吧。自然也不会有符合单一职责这一说法。
  2.要保障类或方法单一职责，并不总是单纯分割代码。更常见的，是要通读逻辑后，通过重构一点一点分离，对原逻辑改动还是比较大的。
  3.我经常说接口要标准化，单一职责就是这里面的一个指标。写新项目可以逼着自己遵守这个。我不接受引入设计原则会降低需求迭代的认知，我相信刻意训练熟能生巧，有序的设计实现功能不会比无厘头的翻译功能慢，更多的是因为手生。但老项目，需不需要改进比如何改进重要。

作者回复2020-07-13 15:31:18

一点一点改动，很有 10x 程序员的味道。